Recollection Haskell, Part III: Recursion

CIS 352/Spring 2015
January 15, 2015

Recursion: Defining something in terms of itself

Question: Who counts as being Jewish?
One answer: Either:
- a. You are Abraham, or
- b. you are a convert, or
- c. your mother was Jewish. (the recursive case)

The Iroquois (Haudenosaunee) have similar rules for clan membership.

The standard first example of recursion: Factorial

```
fact :: Integer -> Integer
fact n
| n==0 = 1
| n>0 = n * fact (n-1)
```

<table>
<thead>
<tr>
<th>n = 4</th>
<th>4 * fact (4-1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>n = 3</td>
<td>4 * 3 * fact (3-1)</td>
</tr>
<tr>
<td>n = 2</td>
<td>4 * 3 * 2 * fact (2-1)</td>
</tr>
<tr>
<td>n = 1</td>
<td>4 * 3 * 2 * 1 * fact 0</td>
</tr>
<tr>
<td>n = 0</td>
<td>4 * 3 * 2 * 1 * 1</td>
</tr>
</tbody>
</table>

The standard 2nd example of recursion: Fibonacci

```
fib :: Integer -> Integer
fib n
| n==0 = 0 |
| n==1 = 1 |
| n>1 = fib(n-1) + fib(n-2) |
| otherwise = error "fib given negative argument" |
```

| n = 0 | 0 |
| n = 1 | 1 |
| n = 2 | fib 1 + fib 0 |
| n = 3 | fib 2 + fib 1 |
| n = 4 | fib 3 + fib 2 |

```
Recursion on lists, 1

Recursions on lists have at least two cases:
1. The list you are recurring on looks like: []
in which case, the recursion bottoms out (i.e., stops).
2. The list you are recurring on looks like: (x:xs)
in which case you probably have subcase where the recursion continues on xs.

    sum' :: (Num a) => [a] -> a
    sum' [] = 0
    sum' (x:xs) = x + sum' xs

A messier example

maximum' :: (Ord a) => [a] -> a
maximum' [] = error "maximum of empty list!"
maximum' [x] = x
maximum' (x:xs) = max x (maximum' xs)

maximum' [2,5,1]
= { (3) succeeds with x = 2, xs = [5,1] } max 2 (maximum' [5,1])
= { (3) succeeds with x = 5, xs = [1] } max 2 (max 5 (maximum' [1]))
= { (2) succeeds with x = 1 } max 2 (max 5 1)
= max 2 5
= 5

Recursion on lists, 3

Class exercises

- replicate' :: Int -> a -> [a]
  replicate' n x = a list of n copies of x
- take' :: Int -> [a] -> [a]
  take' n xs = the first n elements of xs
- reverse' :: [a] -> [a]
  reverse' xs = the reverse of xs
- zip' :: [a] -> [b] -> [(a,b)]
  zip' xs ys = the zip of xs and ys
- elem' :: (Eq a) => a -> [a] -> Bool
elem' x xs tests if x is an element of xs